面向 Open64 的 OpenMP 程序优化①

刘 京, 郑启龙, 李彭勇, 郭连伟

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(中国科学技术大学 安徽省高性能计算重点实验室, 合肥 230027)

摘 要: OpenMP 规范了一系列的编译制导、环境变量和运行库, 具有简单、可移植、支持增量并行等优点. 但同时, 采用 FORK-JOIN 模型所引起的频繁的线程管理开销也是制约 OpenMP 程序性能的瓶颈之一. 本文讨论了如何利用并行区的合并与扩展, 实现并行区的重构, 并在此基础上利用 Open64 的 IPA 优化部件所提供的全局间过程分析能力, 实现跨越过程边界的并行块的合并. 最终实验表明, 该方法有效地改进了 OpenMP 程序的运行性能.

关键词: 增量化并行; 线程管理; IPA; 并行区扩张

OpenMP Program Optimization Based on Open64

LIU Jing, ZHENG Qi-Long, LI Peng-Yong, GUO Lian-Wei

(School of Computer Science and Technology, USTC, Hefei 230027, China)
(Anhui High Performance Computing key laboratory at Hefei, USTC, Hefei 230027, China)

Abstract: OpenMP regulates a series of compilation guidance, environment variables and runtime routines, having the advantages of simple operation, portability and supporting incremental parallel . But at the same time, the use of frequent thread management overhead FORK-JOIN model is one of the bottlenecks caused by OpenMP program performance constraints.. This article discusses how to use the merge and extend parallel zone, reconstruct the parallel section, and on this basis, using global process analysis ability which is provided by IPA, one of the Open64 optimization components, realize the parallel block mergeing which acrosses process boundaries. The final experimental results show that the method improves the performance of OpenMP programs effectively.

Key words: incremental parallelization; thread management; IPA; parallel region expansion

OpenMP[1]是一个达到工业标准的编程 API, 主要适用于共享存储系统. 它提供了一系列的编译制导(Compiler Directive)、运行库(Runtime Library)和环境变量(Environment Variables), 具有简单、移植性好、可扩展性高和支持增量并行化开发等优点.

如今许多编译器都支持 OpenMP,包括一些专用编译器, 诸如 Intel compilers[2], Sun Studio compilers and SGI MIPSpro compilers. 然而, 这些编译器不开放源代码, 从而无法通过这些编译器来获得对 OpenMP 编译技术的理解或者开发一些可能的优化. 有几款开源编译器是可获取的: Omni[3], OdinMP/CCp[4], Mercurium[5]. 这些早期编译器都以 source-to-source 的方式来进行 OpenMP 程序的翻译, 即先将 C 或者 Fortran 语言编写的 OpenMP 程序翻译成各自特定中间语言表示的程序, 该过程实现了 OpenMP 编译制导语义到多线程库调用语义的转变, 然后通过一个 IR-to-source 部件, 将中间语言程序翻译回源程序, 最终该源程序可以在任意支持源语言和多线程库的编译器上运行. 该类编译器在源代码级对 OpenMP 程序的优化通常有如下两种方式: 一是通过确定部分代码的静态词法范围来消除并行域和函数外部的 check 代码, 二是通过仔细分析线程间的数据相关性来消除部分冗余指导语句. 源到源的翻译方式实现简单、易移植, 但是对 OpenMP 程序只具有较小的分析和优化能力.

① 基金项目:"核高基"重大专项(2012ZX01034-001-001)
收稿时间:2015-04-23;收到修改稿时间:2015-05-23

本文采用了 Open64[6]这款编译器来进行 OpenMP 编译优化的研究工作. Open64 是运行在 Linux 下的 C/C++/Fortran 编译基础设施, 设计优良、模块化、健壮, 支持全范围的优化能力. 它有一个叫做 WHIRL 的中间表示语言, 编译器其他部分都在这个中间表示的基础上起作用. 其中, 前端为输入的程序单元生成甚高层 IR, 存储在".B"文件中. 这种中间形式在后续的阶段中都是可用的. 后端可以进一步划分为: 全局优化器、嵌套循环优化器、过程间分析优化器以及代码生成器. 它们工作在不同层次的中间表示级别上. 一个统一的驱动器控制编译器的执行, 确定加载哪个模块以及编译计划. 驱动器还负责模块之间的通信以及各模块的输入管理.

对于 OpenMP 来说, 本文着重关注 Open64 的 IPA 优化部件. 该优化部件能够提供程序的全局过程间调用关系图, 首先传递所有输入文件, 进行符号表解析; 然后进行过程间分析, 如 mod/ref 分析; 再接着进行过程间优化, 读入 WHIRL 并修改它, 如移除函数中某些不可能经过的路径, 最后输出.

1 OpenMP执行模型

OpenMP 使用的是 FORK-JOIN 模型. 该模型如图 1 所示. 从图中可以看出, 程序最初由单独的一个主线程执行. 当遇到由编译制导指示的并行域, 主线程就产生由环境变量指定的多个线程来共同执行并行域内的代码块. 并行域结束时, 只有主线程能穿过并行域继续执行, 其他线程要么被同步, 要么被中断. 主线程创建诸线程的过程称为 FORK, 主线程撤销其他线程的过程称为 JOIN.

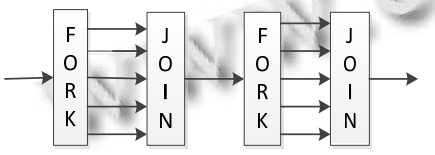

图 1 FORK-JOIN 并行执行模型

该模型有两个特点: 首先它是基于线程的; 其次, 它是一个外部模型, 用户在需要并行的部位插入合适的编译制导语句. 用户可以逐步地(逐个循环地)对串行程序并行化, 得到了增量并行化的好处. 但是同时, FORK-JOIN 是面向任务的模型, 任务和线程之间是动态的关系, 因此存在频繁的线程组产生和撤销操作及时间开销, 其次, 对单个的循环并行化, 妨碍了很多跨循环的优化, 造成了 Cache 命中效率问题[7].

针对 FORK-JOIN 模型的缺点, 本文考虑通过减少并行域的数目, 达到减少线程管理开销和避免妨碍跨循环优化的目的, 从而有效地解决上述问题. 并行区重构的思想是: 若并行域相邻, 在没有数据属性冲突的前提下合并之, 若并行域不相邻, 则扩张并行域. 通过该过程可以使整个程序接近于 SPMD[8]的执行模式, 从而提高程序运行速度.

2 并行区的合并与扩张

2.1 IR-WHIRL

由 Open64 编译器前端生成的 WHIRL 文件包括 WHIRL 指令和 WHIRL 符号表. WHIRL 指令是以程序单元(PU)的形式组织的, 主要用途是代表源代码. 每个 PU 都是单独的一颗树. WHIRL 指令以树状链接, 树中结点叫做 WHIRL 结点. 不允许使用有向无环图. WHIRL 树的另一个作用就是表示程序中的控制关系和表达式语句. 其中有一类结点叫做 REGION, 用来描述并行化的区域. 在编译过程中, 一个并行代码段会被标记为一个并行区域(parallel region).

基于 Open64 中特有的 IR—WHIRL, 可以方便地在其上进行并行区的重构. 算法流程图如下所示:

2.2 并行区的合并

FORK-JOIN 模型的 OpenMP 程序的翻译方案是每当遇到并行域时, 调用运行库提供的线程产生函数, 并将并行块的任务封装在线程的入口函数中, 以此达到并行执行的目标. 每当并行域结束时, 其他线程或被同步或被中断, 只有主线程继续运行. 通过分析该过程可以发现, 若能将相邻的并行域合并起来, 则每合并一次, 就能减少一轮线程创建和撤销的开销. 同时, 在 OpenMP 程序中, 创建并行域的开销远高于并行任务的开销, 两者相差大约 100 倍[9]. 所以并行域的合并能很大地提高 OpenMP 程序的性能.

本文基于 Open64 的中间表示 WHIRL 来对并行区进行合并. 首先遍历 WHIRL 树, 当遇到相邻的并行区时, 就将后一个并行区中的指令全部转移到前一个并行区中, 然后将后一个并行区所在的子树全部删除, 并将删除并行块的 next 指针域指向的下一个块赋给前一个并行区的 next 指针域, 以保持正确的指令执行顺序. 合并前后的 OpenMP 程序如下所示:

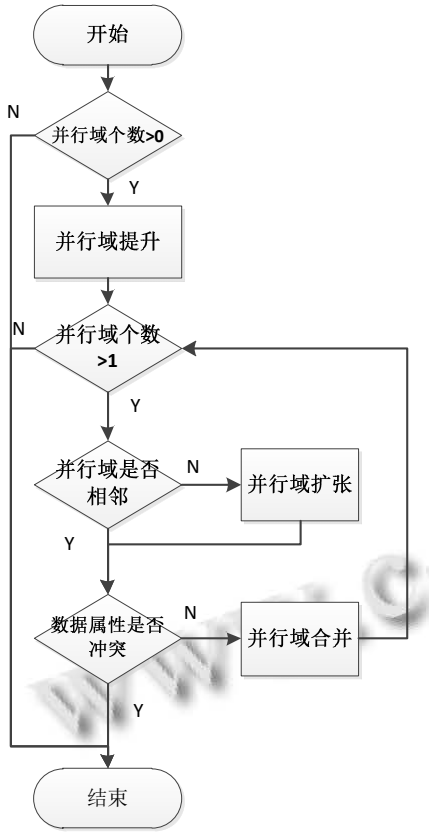

图 2　算法流程图

合并前的 OpenMP 程序:

```
#pragma omp parallel
{
    #pragma omp for
    {
        do_something1();
    }
}
#pragma omp parallel
{
    #pragma omp for
    {
        do_something2();
    }
}
```

合并后的 OpenMP 程序:

```
#pragma omp parallel
{
    #pragma omp for
    {
        do_something1();
    }
    #pragma omp for
    {
        do_something2();
    }
}
```

2.3 并行区的扩张

当两个并行区之间有若干条串行指令时, 无法直接实施并行区的合并. 此时, 可以先进行并行区的扩张, 将串行指令包含进并行区域中, 当两个并行块直接相邻时, 再实施并行块的合并.

并行区的扩张分为左扩张和右扩张. 这里只讨论右扩张的情形, 左扩张形式相同, 操作相反. 操作分三步, 首先将串行指令插入到并行区所在的子树中, 接着对串行指令所在的子树进行变换类型, 以反映其现在已经处于一个并行块之中, 最后删除原来串行指令的一系列节点. 合并前后的 OpenMP 程序如下所示:

扩张前的 Open MP 程序:

```
serial_code;
#pragma omp parallel
{
    do_something();
}
```

扩张后的 OpenMP 程序:

```
#pragma omp parallel
{
    #pragma omp master
    {
        serial_code;
    }
    do_something();
}
```

2.4 跨越过程边界的并行区重构

前面讨论的变换局限于过程内部, 但是Open64独有的全局间过程间分析优化部件 IPA 能够得到整个程序的过程调用关系图, 利用这点, 可以突破常见编译器优化的局限, 通过全局的过程间分析能够知道程序中哪些函数使用了被调函数, 从而就可以通过在使用被调函数的位置处放置合适的编译制导语句来完成跨

越过程边界的并行区重构.

这样做的好处至少有两点: 进一步增大并行块的范围, 从而增加计算的粒度; 同时, 将并行块提升到调用函数中, 可以进一步对调用函数中的并行块合并, 从而使进一步改善程序的结构成为可能. 变换前后的程序如下所示:

变换前的程序:

```
my_procedure
sub_procedure
end
sub_procedure
omp parallel
P
end parallel
end
```

变换后的程序:

```
my_procedure
omp parallel
sub_procedure
end parallel
end
sub_procedure
P
End
```

3 需要注意的问题及解决办法

在上述过程中, 为了成功实现并行域的合并与扩张, 需要注意并解决如下几个问题: ①控制流的处理; ②保证同步行为不被破坏; ③串行代码的保护; ④确定并行域合并的边界; ⑤变量数据属性冲突问题

3.1 控制流的处理

上述并行区的合并与扩张针对的是 WHIRL 树中同一层次的节点, 若并行域在选择或者循环结构的内部, 则需要在纵向上对 WHIRL 树进行修改. 对于 if-else 类型的结构, 若并行块位于 if-else 的 block 区域, 就将它外提, 使得整个 if-else 结构被包含在并行块的内部; 同理, 对于循环结构也进行类似的处理. 整个过程如下所示:

并行块扩张前:

```
if condition then
    omp parallel
    omp end parallel
else
    omp parallel
    omp end parallel
end if
```

并行块扩张后:

```
omp parallel
if condition then
else
end if
omp end parallel
```

3.2 同步处理

无论并行块的合并还是并行块的扩张, 由于并行块的末尾有隐含同步操作, 但是因合并之后并行域的边界撤销从而丢失了一些路障同步, 为了保持程序的正确性以及上下块间原来的存储器视图一致性, 需要添加一些同步语句.

在并行块合并中, 在合并后的并行块间加入 barrier 操作. 在并行块扩张中, 如果是左扩张, 因为无论 master 还是 single 结构, 在结尾处都有一个隐含的 barrier, 所以无需额外的同步语句. 若是右扩张, 则需要在变换后的串行代码区域前加入 barrier.

3.3 串行代码的保护

相邻并行域之间的串行代码因并行域的合并而在多个线程中并发执行, 从而可能引发问题. 可以用 master 或者 single 制导指令来保持其原来的语义.

3.4 确定并行域合并的边界

在前面的讨论中, 本文通过将并行结构从选择语句的分支扩张到整个选择结构, 从循环语句的内部提出到循环外部以及借助于 IPA 实现跨越过程边界的并行块重构. 之所以这样做都基于一个缺省的假设: 尽可能地扩张并行域可以减少产生线程和撤销线程的开销, 从而提升程序的性能. 从原理上说, 可以将 OpenMP 程序全部改造成只带有一个 parallel 制导指令的 SPMD 类型的程序.

虽然大多数情况下 SPMD 类型的程序具有最佳的性能, 但是仔细分析其原因, 可以发现这种优势主要是通过减少线程间的交互, 从而提高 Cache 命中率. 所以本文在进行并行域的重构时, 必须考虑并行块合并的边界. 另一方面, 精确的定量分析 OpenMP 程序性能与并行域边界大小的关系是相当困难的. 基于以

上分析, 本文采用如下几条启发式规则来控制并行域重构算法中的边界.

规则一. 对于控制流中的选择结构和循环结构, 我们总是将分支或者循环内部的并行块扩张到结构之外, 并且重复该过程, 直至并行块包含了整个结构;

规则二. 在一颗 WHIRL 树中, 也就是过程内, 若两个并行域直接相邻, 则合并之; 若两个并行域之间有一个串行节点(最多也只能有一个), 则先将串行节点插入到第一个并行域中, 实施并行域的扩张, 再进行并行域的合并

规则三. 将编译制导语句从被调函数内提升到调用函数中的前提条件是该过程内的所有指令处于一个并行域内.

规则四. 合并过程中遇到无法解决的数据属性冲突或者树的第一层节点中没有其他的并行块, 则对该过程的合并结束.

3.5 变量数据属性冲突问题

并行域合并过程中还必须解决变量数据属性冲突的问题. 考查下面的例子, 同一个变量在并行域 P1 中说明为共享类型, 而在紧接着的并行域 P2 中被说明为私有类型. 这种情况下, 就不能简单地合并 P1 和 P2, 而必须解决该变量的属性冲突. 本文采取的办法利用 OpenMP 的数据属性子句 firstprivate 把共享类型的变量私有化, 前提是在该并行块内没有对该共享变量的定值, 否则无法将该变量的新定值反映到后续的并行块中, 也就无法保证程序的正确性. 数据属性的冲突也是本文所讨论的方法中阻碍并行块合并的唯一因素.

4 实验分析与验证

为了对上文所讨论的并行区优化算法的性能进行分析和验证, 本文使用了测试集 NPB(NSA Parallel Benchmark)[10], 其由日本筑波大学 HPCS 开发, 包含五个核心(CG,MG,FT,IS,EP)和三个应用(BT,SP,LU).

下面给出两个图来说明并行区重构算法的效果, 图 2 是通过该优化算法, OpenMP 程序减少的并行区数目占总数目的比值, 图 3 是优化前后 OpenMP 程序的性能对比.

从图 2 中可以看出, 该优化算法对于减少并行块的数目取得了较好的效果, 能够减少平均 20%的并行块, 其中由于 CG 程序是使用一个共轭梯度方法来计算一个大型稀疏非结构化的矩阵的最小特征值的近似值, 计算比较集中, 所以合并效果相当好.

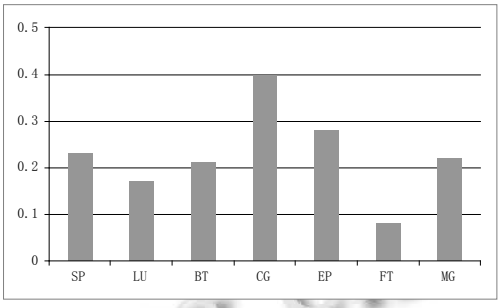

图 2 优化前后并行区数目的对比

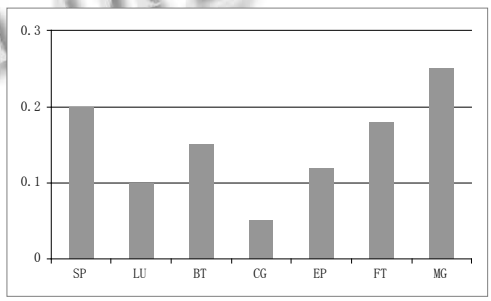

图 3 优化前后 OpenMP 程序的性能对比

图 3 所示是并行块合并后 OpenMP 程序的性能相对于合并之前性能的改变百分比. 从图中可以看出, 程序的性能通过并行区的重构得以提升, 这主要得益于减少线程产生和撤销所节约的开销. 同时, 不难发现, CG 程序的性能提升最为有限. 这是因为在并行区优化算法中, 为了合并不相邻的并行区, 需要进行并行区的扩张, 即将串行代码包含到并行域内, 在此过程中通过添加 barrier 同步来对串行代码进行保护, 从而引入了额外的同步开销, 导致性能提升有限.

5 总结

本文在 Open64 这一开源编译基础设施上, 进行 OpenMP 程序的性能优化. 通过过程内并行域的合并与扩张, 减少 FORK-JOIN 模型下诸线程的产生和撤销开销, 进而提高了程序的运行速度. 并且利用 Open64 独有的全局过程间分析模块, 实现了跨越过程边界的并行块的重构, 这样做的好处至少有两点: 进一步增大并行块的范围, 从而增加了计算的粒度; 通过将并行块提升到调用函数中, 可以进一步对调用中的并行块进行合并, 这样使进一步改善程序的结构成为可能. 但是影响 OpenMP 程序性能的因素颇多, 还有诸如任

务调度开销与负载均衡的关系, 冗余同步的删除等, 这些都是制约 OpenMP 程序性能的重要瓶颈, 也是接下来重点研究的内容.

参考文献

1 Chapman B, Jost G, Van Der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2008.

2 Intel C, User C. Reference Guides. The Intel Compiler Homepage. http://software.intel.com/en-us/intel-compilers, 2009.

3 Sato M, Satoh S, Kusano K, et al. Design of OpenMP compiler for an SMP cluster. Proc. of the 1st European Workshop on OpenMP. 1999. 32–39.

4 Dimakopoulos VV, Georgopoulos A. The OMPi OpenMP/C Compiler. Proc. PCI2005, 10th Panhellenic Conference on Informatics. Volos, Greece. 2005. 153–162.

5 Balart J, Duran A, Gonzàlez M, et al. Nanos mercurium: A research compiler for openmp. Proc. of the European Workshop on OpenMP. 2004, 8.

6 Murphy M. NVIDIA's experience with Open64. Open64 Workshop at CGO. 2008, 8.

7 Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization(CGO). IEEE. 2004. 75–86.

8 Darema F. The spmd model: Past, present and future. Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg. 2001. 1–1.

9 Zhu W, Del Cuvillo J, Gao GR. Performance characteristics of OpenMP language constructs on a many-core-on-a-chip architecture. OpenMP Shared Memory Parallel Programming. Springer Berlin Heidelberg, 2008: 230–241.

10 Van der Wijngaart RF, Wong P. NAS parallel benchmarks version 2.4[Technical Report]. NAS, NAS-02-007, 2002.